

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C10

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade - recap

Functor, Applicative, Monad

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b

class Applicative m => Monad m where
    (=>=) :: m a -> (a -> m b) -> m b
    (=>) :: m a -> m b -> m b
    return :: a -> m a

ma >> mb = ma >=> \_ -> mb
```

Notația **do** pentru monade

($>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do
$e >= \lambda x \rightarrow \text{rest}$	$x <- e$ rest
$e >= \lambda_{} \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

De exemplu

$e1 >= \lambda x1 \rightarrow e2 >> e3$

devine

do

$x1 <- e1$
 $e2$
 $e3$

Exemple de efecte laterale

I/O	Monada IO
Parțialitate	Monada Maybe
Excepții	Monada Either
Nedeterminism	Monada <code>[]</code> (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Monada List

Monada Listă (nedeterminism)

O computație care întoarce un rezultat nedeterminist nu este o funcție pură.

Poate fi transformată într-o funcție pură transformând rezultatul său din tipul a în tipul $[a]$.

În esență, construim o funcție care returnează toate rezultatele posibile în același timp.

Monada Listă (nedeterminism)

Funcțiile din clasa **Monad** specialized pentru liste:

```
(>=>) :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

```
instance Monad [] where
  return x = [x]
  xs >=> f = concat (map f xs)
```

```
concat :: [[a]] -> [a]
```

Monada Listă – exemplu

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
    x <- xs
    if even x
        then [x*x, x*x]
        else [x*x]
```

Monada Maybe

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a

instance Monad Maybe where
    return = Just

    Just va >>= f = f va
    Nothing >>= _ = Nothing
```

Monada Maybe – exemplu

```
radical :: Float -> Maybe Float
radical x
| x >= 0 = return (sqrt x)
| x < 0  = Nothing

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return (negate c / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Maybe – exemplu

```
-- a * x^2 + b * x + c = 0
solEq2All :: Float -> Float -> Float -> Maybe [Float]
solEq2All 0 0 0 = return [0]
solEq2All 0 0 c = Nothing
solEq2All 0 b c = return [negate c / b]
solEq2All a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    let s1 = (negate b + rDelta) / (2 * a)
    let s2 = (negate b - rDelta) / (2 * a)
    return [s1,s2]
```

Monada Either

Monada Either (a exceptiilor)

```
data Either err a = Left err | Right a

(>>=) :: Either err a -> (a -> Either err b) ->
            Either err b
return :: a -> Either err a

instance Monad (Either err) where
    return = Right

    Right va >>= f = f va
    err >>= _ = err
-- Left verr >>= _ = Left verr
```

Monada Either – exemplu

```
radical :: Float -> Either String Float
radical x
| x >= 0 = return (sqrt x)
| x < 0  = Left "radical:„argument„negativ"

-- a * x^2 + b * x + c = 0
soleEq2 :: Float -> Float -> Float -> Either String Float
soleEq2 0 0 0 = return 0
soleEq2 0 0 c = Left "ecuatie:„fara„solutie"
soleEq2 0 b c = return (negate c / b)
soleEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Writer

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip
```

Funcțiile din clasa **Monad** specializezate pentru Writer:

```
(>>=) :: Writer log a -> (a -> Writer log b) ->
           Writer log b
return :: a -> Writer log a
```

```
instance Monad (Writer String) where
    return va = Writer (va, "")
    ma >>= f = let (va, log1) = runWriter ma
               (vb, log2) = runWriter (f va)
               in Writer (vb, log1 ++ log2)
```

Monada Writer - exemplu

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```

```
tell :: log -> Writer log ()  
tell msg = Writer ((), msg)
```

```
logIncrement :: Int -> Writer String Int  
logIncrement x = do  
    tell ("increment:" ++ show x ++ "--")  
    return (x + 1)
```

```
logIncrement2 :: Int -> Writer String Int  
logIncrement2 x = do  
    y <- logIncrement x  
    logIncrement y
```

```
> runWriter (logIncrement2 13)  
(15,"increment:13--increment:14")
```

Monada Reader

Monada Reader (stare nemodificabilă)

```
newtype Reader env a = Reader {runReader :: env -> a}
-- runReader :: Reader env a -> env -> a

instance Monad (Reader env) where
  return = Reader const
  -- return x = Reader (λ_ -> x)

  ma >>= k = Reader f
    where
      f env = let va = runReader ma env
              in runReader (k va) env
```

Monada Reader - exemplu

```
newtype Reader env a = Reader {runReader :: env -> a}
-- runReader :: Reader env a -> env -> a

ask :: Reader env env
ask = Reader id

tom :: Reader String String
tom = do
  env <- ask -- gives the environment (here a String)
  return (env ++ "This is Tom.")

jerry :: Reader String String
jerry = do
  env <- ask
  return (env ++ "This is Jerry.")
```

Monada Reader - exemplu

```
tomAndJerry :: Reader String String
tomAndJerry = do
    t <- tom
    j <- jerry
    return (t ++ "\n" ++ j)

runJerryRun :: String
runJerryRun = runReader tomAndJerry "Who is this?"
```

Exercițiu

Se dă tipul de date

```
data Fuel a = Fuel {getFuel :: Integer -> Maybe (Integer , a)}
```

Să se scrie instanța clasei **Monad** pentru tipul **Fuel**, ținând cont că în timpul computațiilor se consumă câte o unitate din resursă (**fuel**), iar computațiile pot fi efectuate cât timp resursa nu este 0.

Exercițiu

Exemplu:

```
test :: Fuel Int
test = pure 0 >>= pure . (+1) >>= pure . (+1)

failEx = getFuel test 1          -- Nothing

sucEx = getFuel test 2          -- Just (0,2)
```

Exercițiu

```
instance Functor Fuel where
    fmap f (Fuel m) =
        Fuel (\i -> case m i of
            Nothing -> Nothing
            Just (i', a) -> Just (i',
                f a)
        )
```

Exercițiu

```
instance Applicative Fuel where
    pure x = Fuel (\i -> Just (i, x))
    (Fuel mf) <*> (Fuel af) =
        Fuel (\i -> case mf i of
            Nothing -> Nothing
            Just (i', f) ->
                if i' <= 0
                    then Nothing
                else f (Fuel af))
```

Exercițiu

```
else case af (i' - 1) of
    Nothing -> Nothing
    Just (i'', a) ->
        if i'' <= 0
        then Nothing
        else Just (i'' - 2, f a)
)
```

Succes la examen!